Accelerating DCA++ (Dynamical Cluster Approximation) Scientific Application on the Summit supercomputer

Giovanni Balduzzi[¶]
ETH Zurich
8093 Zurich, Switzerland
gbalduzz@itp.phys.ethz.ch

Arghya Chatterjee[¶]
Oak Ridge National Laboratory
Oak Ridge 37831, U.S.
chatterjeea@ornl.gov

Ying Wai Li[†¶]
Los Alamos National Laboratory
Los Alamos 87545, U.S.
yingwaili@lanl.gov

Peter W. Doak
Oak Ridge National Laboratory
Oak Ridge 37831, U.S.
doakpw@ornl.gov

Urs Haehner
ETH Zurich
8093 Zurich, Switzerland
haehneru@itp.phys.ethz.ch

Ed F. D'Azevedo
Oak Ridge National Laboratory
Oak Ridge 37831, U.S.
dazevedoef@ornl.gov

Thomas A. Maier
Oak Ridge National Laboratory
Oak Ridge 37831, U.S.
maierta@ornl.gov

Thomas Schulthess[‡‡]
ETH Zurich
8093 Zurich, Switzerland
schulthess@cscs.ch

***Abstract*—Optimizing scientific applications on today's accelerator-based high performance computing systems can be challenging, especially when multiple GPUs and CPUs with heterogeneous memories and persistent non-volatile memories are present. An example is Summit, an accelerator-based system at the Oak Ridge Leadership Computing Facility (OLCF) that is rated as the world's fastest supercomputer to-date. New strategies are thus needed to expose the parallelism in legacy applications, while being amenable to efficient mapping to the underlying architecture.**

In this paper we discuss our experiences and strategies to port a scientific application, DCA++, to Summit. DCA++ is a high-performance research application that solves quantum many-body problems with a cutting edge quantum cluster algorithm, the dynamical cluster approximation.

Our strategies aim to synergize the strengths of the different programming models in the code. These include: (a) streamlining the interactions between the CPU threads and the GPUs, (b) implementing computing kernels on the GPUs and decreasing CPU-GPU memory transfers, (c) allowing asynchronous GPU communications, and (d) increasing compute intensity by combining linear algebraic operations.

Full-scale production runs using all 4600 Summit nodes attained a peak performance of 73.5 PFLOPS with a mixed precision implementation. We observed a perfect strong and weak scaling for the quantum Monte Carlo solver in DCA++, while encountering about 2× input/output (I/O) and MPI communication overhead on the time-to-solution for the full machine run. Our hardware agnostic optimizations are designed to alleviate the communication and I/O challenges observed, while improving the compute intensity and obtaining optimal performance on a complex, hybrid architecture like Summit.

***Index Terms*—DCA, Quantum Monte Carlo, QMC, CUDA, CUDA aware MPI, Summit@OLCF, Spectrum MPI**

I. Introduction

With rapidly changing microprocessor designs, the next generation high performance computers will have massive amounts of hierarchical memory available on the node, from user-managed caches, DRAM, high bandwidth memory and non-volatile memory (NVMs). On the other hand, exploring the best ways to integrate multiple programming models for collective optimization of performance remains one of the biggest challenges. Systems like OLCF's Summit[1] generate the majority of floating point operations per second (FLOPS) from GPUs (as high as 97%), which are throughput optimized with a large number of threads, while CPUs are latency optimized. Thus, finding the right balance between memory access and computation work distribution, according to the underlying heterogeneous hardware without sacrificing efficiency and scalability, is still an outstanding research problem.

The first step to addressing this issue is to understand how "asynchronous" execution models can be used to: (a) describe units of work; (b) let the runtime system efficiently schedule work to hide the latency in accessing various memory hierarchies and NUMA domains; and (c) determine if heterogeneous threads are a possible solution. Current programming models that provide "rigid" synchronization constructs (e.g., full system barriers at the end of loops) may not be able to scale due to the high overhead of the massive parallelism with threads.

[¶]All three authors have equal contribution to the manuscript.

† Ying Wai Li contributed to this work mostly during her previous appointment at Oak Ridge National Laboratory, Oak Ridge 37831, U.S.

[‡‡]Thomas Schulthess is also affiliated with Swiss National Supercomputing Center (CSCS), 6900 Lugano, Switzerland.

[1]World's fastest supercomputer with a theoretical performance of 200 peta-FLOPS (PFLOPS) as of June 2019 [1].

In this work, we discuss the programming styles and strategies that were used in the development of DCA++. DCA++ implements a quantum cluster method [2] known as the dynamical cluster approximation (DCA) [2]–[4], with a quantum Monte Carlo (QMC) kernel for modeling strongly correlated electron systems. The DCA++ code currently uses three different programming models (MPI, CUDA, and C++ Standard threads), together with numerical libraries (BLAS, LAPACK and MAGMA), to expose the parallelization in computations. Optimizing one single programming model does not necessarily help us achieve our efficiency / scalability goal. We discuss our vision on how we may use a general programming style or strategy(ies) to exploit the underlying memory hierarchy.

A. Contribution

The primary contributions of this work are outlined below:

(a) Custom thread pool for on-node parallelization and managing GPU driver threads.
(b) Improved delayed sub-matrix updates to increase GPU utilization.
(c) Asynchronous communication among multiple GPUs on Summit.
(d) Accumulation of complex measurements ported from the CPU to GPU, accounting for significant performance improvement in runtime and GPU utilization.
(e) Delayed Non-uniform Fast Fourier Transform in single-particle accumulation.
(f) Batched matrix-matrix multiplication for small 2D Fourier transforms in two-particle accumulation.
(g) Mixed precision operations on GPUs.

II. Background

The study and understanding of strongly correlated electronic systems is one of the greatest challenges in condensed matter physics today. Strongly correlated electron systems are characterized by strong electron-electron interactions, which give rise to exotic states and fascinating properties such as multiferroicity[2] and high-temperature superconductivity. These properties open the door for technological advances in applications such as data storage and MRI machines. However, current methodologies based on density functional theory (DFT), the workhorses for electronic structure calculations, fail to describe the effects of correlations that govern the physics in strongly interacting electron systems.

The complexity of the general electronic structure problem and the failure of DFT has led to the development of reduced models that are believed to capture the relevant physics underlying the observed properties. The most prominent example is the use of the two-dimensional (2D) Hubbard model for the study of high-temperature superconducting copper-oxide based materials (cuprates) [5], [6], for instance. The Hubbard model describes interacting electrons on a lattice, which can hop between lattice sites and interact through an on-site Coulomb repulsion. Formally, the Hamiltonian is given by

$$H = H_0 + H_{\text{int}} = -t \sum_{\langle i,j\rangle,\sigma} c^\dagger_{i\sigma} c_{j\sigma} + U \sum_i n_{i\uparrow} n_{i\downarrow} \,. \qquad (1)$$

The first term of (1), where $\langle i, j\rangle$ indicates that the sum runs over nearest-neighbor sites i and j, represents the electron hopping with amplitude t. The second term, where the sum runs over all lattice sites i, captures the on-site Coulomb interaction of strength U. The index $\sigma \in \{\uparrow, \downarrow\}$ represents the electron spin. Systems with multiple electron bands per lattice site are also supported.

Mathematically, the Hamiltonian is represented by a matrix. Solving for the possible energy levels and states of the system is equivalent to solving for the eigenvalues and eigenstates of the Hamiltonian matrix. However, exact diagonalization studies of the 2D Hubbard model are restricted to very small lattices as the problem size scales exponentially with the number of lattice sites. Quantum Monte Carlo simulations, in turn, are plagued by the infamous fermion sign problem, which again limits the accessible lattice sizes and prevents calculations at low temperatures [7]. To study Hubbard model type of problems, dynamical mean field theory (DMFT) [8] has become one method of choice. In DMFT the complexity of the infinite lattice problem is reduced by mapping it to a self-consistent solution of an effective impurity model, thereby treating only spatially local correlations exactly and including non-local correlations on a mean-field level. In order to treat additional non-local correlations that are important to understand the mechanism of high-temperature superconductivity, for example, DMFT has been extended by quantum cluster approaches such as the dynamical cluster approximation (DCA) [2]–[4].

DCA is a numerical simulation tool to predict physical behaviors (such as superconductivity, magnetism, etc.) of correlated quantum materials [9]. The DCA++ code[3] computes the many-body (single-particle and two-particle) Green's functions for a Hubbard-type material of interest. Properties of the materials, such as the superconducting transition temperature, can be calculated from these many-body Green's functions.

DCA++ has an iterative, self-consistent algorithm with two primary kernels (see Fig. 1): (a) Coarse-graining of the single-particle Green's function to reduce the complexity of the infinite size lattice problem to that of an effective finite size cluster problem, and, (b) quantum Monte Carlo based solution of the effective cluster problem.

Almost all of DCA's computing time is spent in the QMC solver. Fig. 2 shows its general workflow.

A. Coarse-graining

The DCA algorithm replaces the infinite lattice problem by a finite-size impurity cluster that is embedded in a

[2]Multiferroics are materials exhibiting more than one ferroic property such as ferromagnetism and /or ferroelectricity.

[3]The DCA++ code has been created in a collaboration between Oak Ridge National Laboratory (ORNL) and ETH Zurich. ORNL's DCA++ code won the Gordon Bell Award in 2008 for the first petascale computation of high-temperature superconductors [10].

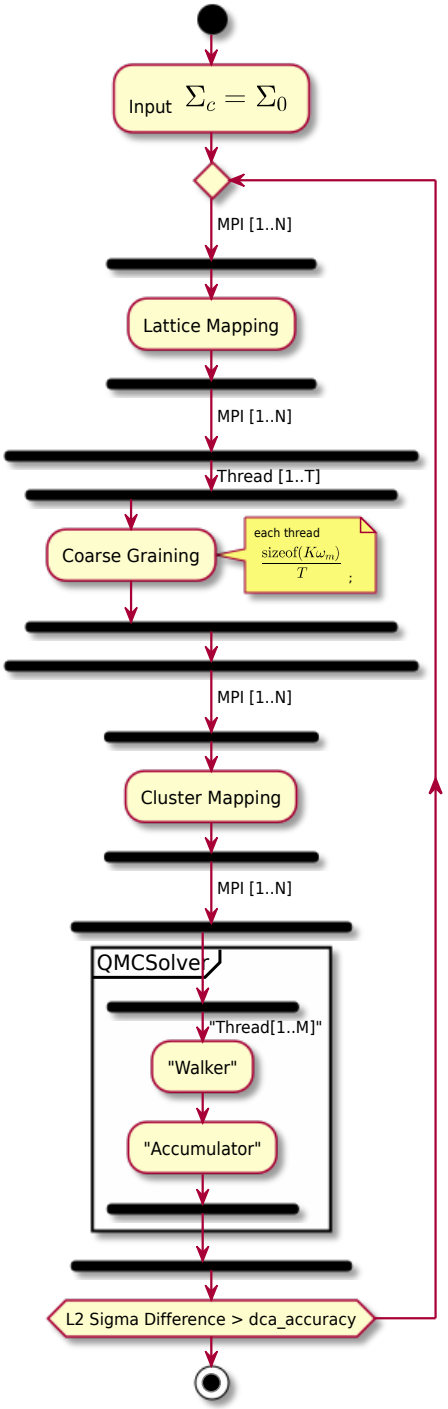

Fig. 1. General workflow of the DCA++ application, showing two primary kernels and input/output to each of the kernels. On distributed multi-core machines we exploit the underlying hardware with a two level (MPI + threading) parallelization scheme.

self-consistent mean-field. Formally, we substitute the lattice self-energy by $\Sigma_{\text{DCA}}(\vec{k}, i\omega_n)$, a piecewise constant continuation of the cluster self-energy $\Sigma_c(\vec{K}, i\omega_n)$:

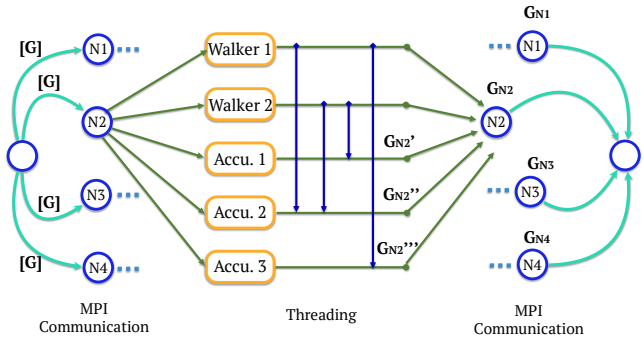

Fig. 2. The computation structure of the quantum Monte Carlo kernel. This figure shows the levels of parallelism in an iteration of the QMC solver (the MPI communication, thread level parallelism and accelerator level). Each rank (N1 – N4) is assigned a Markov Chain and the initial Green's function, G. Each rank spawns worker threads (walkers and accumulators). Computation performed by the walker threads are done on the GPU, upon completion the walkers send measurements over to the accumulator threads, running on the CPU (asynchronous computation) to generate partial G's (G_{N2}', G_{N2}'', G_{N2}''' etc. on node N2). The partial G's are then reduced within the node to give G_{N2} (G_{N1}, G_{N3} and G_{N4} for nodes N1, N3 and N4 accordingly), followed by a `MPI_AllReduce` operation that computes the final G. This G is then fed into the coarse-graining step in the next iteration.

$$\Sigma_{\text{DCA}}(\vec{k}, i\omega_n) = \sum_{\vec{K}} \phi_{\vec{K}}(\vec{k})\, \Sigma_{\text{c}}(\vec{K}, i\omega_n)\,, \tag{2}$$

where $\omega_n = \frac{2n\pi}{\beta}$ is a Matsubara frequency, $\beta = 1/T$ is the inverse temperature, and the patch function $\phi_{\vec{K}}(\vec{k})$ is one if the momentum $\vec{k}$ lies inside the $\vec{K}^{\text{th}}$ patch, and is zero otherwise. The sum runs over a mesh of points in $\vec{K}$-space, centered around by the reciprocal cluster points $\vec{k}$.

The single-particle Green's function $G(\vec{k}, i\omega_n)$, which describes the propagation of a single electron bearing a momentum $\vec{k}$ and a frequency ω_n, is then coarse-grained over the patches to obtain the coarse-grained single-particle Green's function $\bar{G}(\vec{K}, i\omega_n)$:

$$\bar{G}(\vec{K}, i\omega_n) = \tfrac{N_c}{V_{\text{BZ}}} \int d\vec{k}\, \phi_{\vec{K}}(\vec{k}) \left[G_0^{-1}(\vec{k}, i\omega_n) - \Sigma_{\text{DCA}}(\vec{k}, i\omega_n) \right]^{-1}, \tag{3}$$

where N_c denotes the cluster size, V_{BZ} is the volume of the Brillouin zone, and

$$G_0(\vec{k}, i\omega_n) = \frac{1}{i\omega_n - H_0(\vec{k}) + \mu} \tag{4}$$

is the Green's function corresponding to the non-interacting part of the Hamiltonian, H_0 in (1). The chemical potential μ is a parameter that needs to be adjusted by iterating the coarse-graining step until the desired electron density is obtained. The density can be calculated from the value of the Fourier transform of $G(\vec{k}, i\omega_n)$ into real space and time coordinates, then evaluated at zero time and displacement in real space.

This makes the problem tractable by reducing the degrees of freedom to those of the cluster, while still retaining information about the remaining lattice degrees of freedom in an averaged fashion. The coarse-grained Green's function $\bar{G}(\vec{K}, i\omega_n)$

and the cluster self-energy $\Sigma_c(\vec{K}, i\omega_n) = \Sigma_{\text{DCA}}(\vec{K}, i\omega_n)$ define the bare Green's function of an effective cluster problem through the Dyson equation,

$$\mathcal{G}_0(\vec{K}, i\omega_n) = \left[\bar{G}^{-1}(\vec{K}, i\omega_n) + \Sigma_c(\vec{K}, i\omega_n)\right]^{-1}. \quad (5)$$

From $\mathcal{G}_0(\vec{K}, i\omega_n)$, we can use the Monte Carlo techniques, to be described in the following subsection, to produce a corrected coarse-grained single-particle Green's function that includes the contribution from the interacting part of the Hamiltonian, H_{int} in (1). We can then calculate a new cluster self-energy $\Sigma_c(\vec{K}, i\omega_n)$ for the next iteration, which closes the DCA self-consistency loop.

B. Quantum Monte Carlo (QMC) solver

We employ a continuous time auxiliary-field (CT-AUX) QMC algorithm [11], [12]. Our implementation of the CT-AUX solver incorporates submatrix updates [12] and accumulation of measurements with non-equidistant fast Fourier transforms [13]. In the CT-AUX methodology, the partition function Z is expressed as an expansion in terms of the N-matrices $\left(N^{\sigma}_{\{s_i,\tau_i\}_k}\right)$:

$$Z = \sum_{k\geq 0} \sum_{\substack{s_i=\pm 1 \\ 1\leq i\leq k}} \int_0^\beta d\tau_1 \cdots \int_{\tau_{k-1}}^\beta d\tau_k \left(\frac{K}{2\beta}\right)^k Z_k(\{s_i,\tau_i\}_k),$$
$$Z_k(\{s_i,\tau_i\}_k) = Z_0 \prod_{\sigma=\uparrow,\downarrow} \left|N^{\sigma}_{\{s_i,\tau_i\}_k}\right|^{-1}, \quad (6)$$

and $Z_0 = \text{Tr}e^{-\beta H_0}$. Here, the imaginary time $\tau_1 \ldots \tau_k \in [0, \beta)$. The constant K is introduced to express Z in the interacting representation; the quartic interaction term in the Hamiltonian has been replaced through a Rombout's decoupling by a coupling to an Ising auxiliary spin field s_i [11], and $\sigma = \uparrow, \downarrow$ is the electron spin. The outer sum runs over the expansion order k, and the inner sum runs over the k auxiliary spin fields s_i.

During the simulation, the expansion order k has a peaked distribution around a mean value that scales as $O(N_c\, U/\, T)$, where N_c is the cluster size and T is the temperature. The memory requirement for the random walk scales as $O(k^2)$, while the computational cost scales as $O(k^3)$.

The quantity that needs to be measured from the random walk (by the walkers), and to be used in the following iterations of the DCA loop, is the single-particle Green's function (dropping variable dependence for clarity):

$$G = \mathcal{G}_0 - \mathcal{G}_0\, M\, \mathcal{G}_0, \quad (7)$$

where $G = G(\vec{k}, i\omega_n)$ and $\mathcal{G}_0 = \mathcal{G}_0(\vec{K}, i\omega_n)$ are introduced in Section II-A, M is a $k \times k$ matrix produced by the MC walker, closely related to the inverse of N from (6).

Samples of M, which is a translational invariant function of real space and imaginary time, are measured during the MC procedure. Working in frequency and momentum Fourier space has the advantage that all terms that enter in (7) are diagonal. We therefore perform a series of batched 2D Fourier transforms over the two time indices of M to transform them to the frequency domain. Similarly, we transform the space indices to the momentum domain.

Besides the single-particle function, another important quantity to accumulate is the 4-point two-particle correlation function G_{tp}. We compute G_{tp} in the particle-particle channel, from which one can extract information of the superconducting behavior. In the QMC algorithm, this function is accumulated according to

$$G_{\text{tp}}(K_1, K_2, K_3) \mathrel{+}= \sum_{\sigma} G_{\sigma}(K_3 - K_2, K_3 - K_1)\, G_{-\sigma}(K_2, K_1)\,. \quad (8)$$

Here K_i is a combined index representing a particular point in the momentum and frequency space, and σ specifies the electron spin value.

This G_{tp} correlation function describes the propagation of a pair of electrons with opposite spin from momentum/frequency $(K_2, K_3 - K_2)$ and center of mass momentum/frequency K_3 to their scattered states with momentum/frequency $(K_1, K_3 - K_1)$.

The Appendix summarizes the definition of each physical quantity introduced in this section and to be used in the the next section for the implementation of DCA++ algorithm.

III. Implementation

In this section, we outline our implementation of the improved DCA++ code, as well as our design and parallelization strategies on the Summit supercomputer. As shown in Fig. 1, the two main kernels of DCA++ are: (a) the coarse-graining step, and, (b) the quantum Monte Carlo solver. A primary challenge in the optimization of this code is that the runtime of different sections of the algorithm scales differently with a subset of the input parameters.

A. Input parameters

We set up a low temperature simulation of a system representing a typical production level execution: a single-band square lattice Hubbard model with Coulomb repulsion $U/t = 4$, temperature $T/t = 0.02$ and DCA cluster size $N_c = 36$. The average expansion order observed with this set of parameters is $k = 2600$. A fixed number of 80 million total measurements were performed to study strong scaling, while a fixed number of 50,000 measurements per node (i.e., a fixed runtime of about 12 minutes) were performed for the weak scaling study.

Each simulation sampled $\omega_{\text{sp}} = 2048$ Matsubara frequencies for the single-particle Green's function and $\omega_{\text{tp}} = 128$ Matsubara frequencies for each dimension of the two-particle Green's function. All the 10 possible independent momentum transfers were computed, while the exchange frequency was fixed at $\Delta\omega = 0$. The performance evaluation, resource allocation, and problem specific optimization strategies reported later in Sections III and IV will refer to this system.

B. Coarse-graining

The coarse-graining step in (3) does not scale with the inverse temperature, and its contribution to the runtime is generally negligible for a production level run. Nevertheless its overhead can impact the testing and development of the code, especially as, depending on the initial configuration, (3) needs to be recomputed several times until the electron density has converged to the target density.

We parallelized this step of the algorithm with a combination of MPI distribution and multi-threading (C++ std::threads). Each thread of each process can simply integrate the Green's function independently for a set of frequency and cluster momentum values.

This portion of the code greatly benefited from our new parallelization implementation based on a thread pool, described in more detail in Section III-D. We further optimized the code by: (a) improving the scheduling of the intra-process work, (b) implementing specialized functions for the inversion of small matrices, and (c) reducing the number of floating point operations by taking advantage of the spin symmetry in (3). Note that the symmetric property of an Hamiltonian matrix also reduces the memory overhead.

All these optimization led to a significant speedup of the coarse-graining step compared to the previous DCA++ implementation in version 1.0.0 [14]. The performance varies widely depending on different system sizes and input parameters; on Summit we observed an improvement of $100\times$ to $1000\times$.

C. Quantum Monte Carlo (QMC) solver

The high dimensional integral in (6) is solved by Monte Carlo methods. The Monte Carlo integration in DCA++ is the most computationally intensive step. When porting DCA++ to Summit, this kernel is where we spent most of our optimization effort. These optimizations include the employment of various parallelization techniques, getting higher GPU utilization, reducing memory usage and addressing load imbalance across nodes.

Monte Carlo algorithms can be massively parallelized by executing concurrent, independent Markov chains across multiple nodes. Considering that the majority of the computing capacity of a simultaneous multi-threading (SMT) machine (e.g. Summit) comes from the GPUs, one must optimize these embarrassingly parallel kernels, both inter-node and on-node, to take full advantage of the underlying hardware.

On machines with hybrid nodes consisting of multi-core CPUs and GPUs, Monte Carlo implementation and parallelization strategies are hence a little different. Naive implementation of the QMC kernel is not suitable for GPU acceleration, because the algorithm contains many decision branches based on the stochastic acceptance or refusal of each update of the configuration, which needs to be performed on the CPU. To increase the compute intensity during this Monte Carlo acceptance-rejection step, DCA++ implements the continuous-time auxiliary-field algorithm (CT-AUX) with a submatrix update technique that groups multiple matrix-vector multiplications into a single matrix-matrix multiply [15]. This has already been implemented in version 1.0.0 [14].

*1) **Monte Carlo (MC) walkers**:* In the QMC solver, the MC walks are performed to sample the expansion order (k) space in (6) to obtain an estimation of the sum of the series expansion. Pictorially, one might imagine a MC move as a "spin flip" of the Ising auxiliary spin field s_i, which results in a change in a column of a matrix. The acceptance probability of each MC move is proportional to the ratio of the determinant of the updated matrix with the determinant of the old one (see e.g., [10] for more details). If the explicit inverse of the old matrix is available, the new determinant can be computed using the matrix-determinant lemma, which entails a matrix-vector multiplication of $O(k^2)$ complexity.

In our implementation, the MC walkers combine the effect of several MC moves into a large matrix-matrix multiplication and perform the computation on the GPU [12]. While the complexity of this method is still $O(k^2)$ per update, it ensures that we use the GPU cache efficiently while reducing the rounds of CPU-GPU communication.

The CPU processes the acceptance of each individual move and their interaction with each other by growing a smaller submatrix. We optimized the previous implementation by using asynchronous copies between the CPU and GPU. We also improved the implementation of custom kernels for matrix row and column reorientation by exposing more parallelism to the GPU threads. These optimization in the MC walker kernel led to a performance benefit of $1.3\times$ measured while running a single CPU thread and a single GPU stream.

*2) **Monte Carlo two-particle accumulation**:* The second part of the QMC solver is the measurement of the configurations generated by the MC walkers. We also calculate a larger two-particle correlation function to capture interesting physical phenomena of condensed matter, such as superconducting phase transition at lower temperature.

Based on the input parameters, the number of floating point operations required to accumulate a measurement scales up to $O(k^2\,\omega_{\text{tp}} + k\,\omega_{\text{tp}}^2 + \omega_{\text{tp}}^3)$, where k is the expansion order of the MC integration, and ω_{tp} is the number of discrete Matsubara frequencies stored for each dimension of the two-particle function (see Section III-A for the input parameters we use as our test case). The number of frequencies required to capture a phase transition scales with the inverse temperature, which scales with the expansion order. Hence, the computational cost of the accumulation step becomes as relevant as the MC walk, and efficient utilization of the GPUs on hybrid architectures like Summit is necessary.

Version 2.0 also supports computing multiple frequencies and momentum transfers in a single execution, allowing the computation of the full correlation function in a single run.

Our main challenge while developing version 2.0 was to create a high performance GPU code that also minimizes the amount of data movement between the CPU-GPU and GPU-GPU memory. The memory footprint is of particular importance, as the output correlation function takes more than 3 GB of device memory, while each intermediate result

takes around 750MB of memory for the specific system in Section III-A.

To address this problem, we store intermediate GPU results in private class members of type `std::shared_pointer`. Each C++ objects can be executed and tested independently, while a higher level manager object can direct each implementation object to reuse the same memory allocation, when appropriate.

The two-particle accumulation is implemented by invoking the following kernels on a GPU stream:

(a) **2D transform from time to frequency domain.** The first step is to perform a 2D Fourier transform of M from the time domain to the frequency domain. While this transformation is implemented as a delayed non-uniform fast Fourier transform (DNFFT) in the case of the single-particle function, for the two-particle case we implemented it as a matrix-matrix multiplication.
The non-uniform fast Fourier transform algorithm (NFFT) requires the convolution of the data on a uniform, fine grid before applying the FFT algorithm. The memory required to store a 2D convolution of each orbital pair is too expensive that it is impossible to execute them concurrently with sufficient accuracy, unless the number of transforms is limited. In our test case, for each MC sample, 1296 Fourier transforms of matrices of average size 72×72 are performed. Considering the bandwidth of the global memory and the peak performance of a NVIDIA Tesla V100, the problem is close to the ridge point of a roofline model. Hence matrix-multiplication is favored over NFFT, in terms of global memory consumption and access.
We have also considered using the 3M algorithm [16] to implement the complex matrix multiplications with a reduced number of floating point operations. While we use it in the CPU version, we opted out in the GPU implementation due to additional memory requirement. We used the batched matrix multiplication routine in the MAGMA 2.4.0 library [17] to execute the Fourier transforms in parallel on the GPU.

(b) **2D transform from space to momentum domain.** Another set of 2D Fourier transforms needs to be performed on the space indices of the input tensor. In our test case, 8385 transforms of 36×36 matrices are performed, using batched matrix-matrix multiplications. Using a batched FFT algorithm might be necessary if the DCA++ code was extended to larger clusters, but in our use case the matrix-matrix multiplication was more reasonable. Using a custom Fourier matrix allows us to easily work with clusters of arbitrary shapes and spatial dimensions, and to combine the Fourier transform with another linear transformation used to improve the coarse-graining results in certain cases.

(c) **Single-particle Green's function (G) computation.** The single-particle function G is computed according to (7).

(d) **Two-particle Green's function ($G_{\mathbf{tp}}$) accumulation.** Finally the single-particle functions are combined to give the two-particle function through (8). When more than one accumulation stream per GPU is run concurrently, this update is an atomic operation. This operation is the most memory intensive part of the computation. While storing G_{tp} on the device allows us to exploit the much larger bandwidth of GPU memory compared to RAM and to avoid communications with the CPU, we are limited to the device memory size. Looking forward, a distributed MC sampling scheme will need to be implemented to enable calculations using larger clusters with higher frequency and momentum transfer.

D. Parallelization strategies

Monte Carlo simulations are embarrassingly parallel, which we exploit on distributed multi-core machines with a two level (MPI + threading) parallelization scheme (see Fig. 2). On node, we parallelize the Monte Carlo over several CPU threads using a custom thread pool. We create several instances of independent Markov chains, each managed by a *walker object* (producer), and one or more *accumulator object(s)* (consumer) that measures the single- and two-particle Green's functions. In addition, we employed the following strategies to further improve the performance of our code.

Running multiple *walker* objects concurrently. This helps keep the GPU busy while a memory copy is performed. Each GPU stream associated with the walker waits for the sub-matrix computation performed on the CPU. As a *walker* performs FLOP-intensive operations both on the CPU and on a CUDA stream, we recommend using as many *walkers* as CPU cores, and up to the same number of *accumulators* provided it fits in the GPU memory.

Dynamically distributing work at runtime. Unlike version 1.0.0 of DCA++ where the measurements were statically distributed among MPI ranks and threads, in version 2.0 we addressed the load imbalance across threads, by allowing each *accumulator* to measure a different number of samples, up to a fixed number of measurements per rank. The measurements are still distributed statically over the MPI ranks to avoid inter-node communications, while within the MPI process the measurements are dynamically assigned to idle threads. For walker-accumulator synchronization we use two techniques, either shared threads or separate walker / accumulator thread.

Using equal number of *walkers* and *accumulators*. There is no communication between different CPU threads; after a *walker object* produces a MC sample, the *accumulator* is called immediately, and en-queues the measurement on an independent GPU stream, allowing the *walker* to immediately start another MC step. Fig. 3 shows a UML diagram of the synchronization of the shared *walker* and *accumulator* threads.

Using less *accumulators* than *walkers* due to limitations in memory. Available *accumulators* wait in a queue. Upon the generation of a new MC configuration, the *walker* copies it to the front of the queue and signals the *accumulator* to start the measurement, and resumes the random walk. When the *accumulator* thread finishes its measurement, it goes to the back of the queue. This method was already implemented in

version 1.0.0; we improved the efficiency of the queries to the queue by using the synchronization primitives (`std::mutex` and `std::conditional_variable`) offered by the C++ Standard Library (STL). Fig. 4 shows a UML diagram of the synchronization between the separate *walker* and *accumulator* threads.

Careful memory management when using multiple accumulators. Each accumulator object stores a private copy of the single-particle measurements, while the more memory intensive two-particle results are accumulated atomically to the same address space. We observed no performance benefit by running version 2.0 with more software threads than physical cores, and only negligible gain in using two threads per core for the accumulators in version 1.0.0.

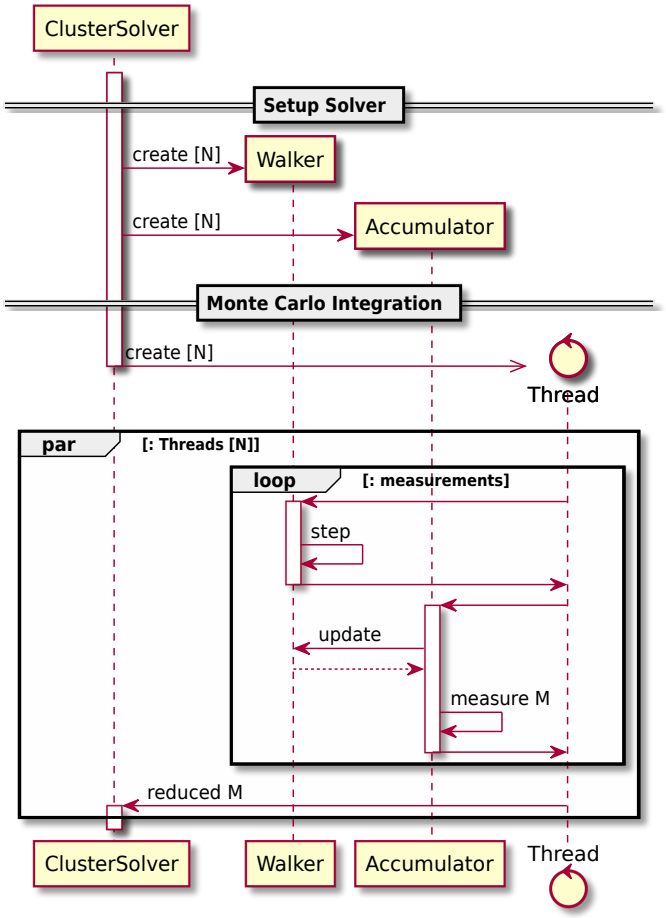

Fig. 3. UML sequence diagram of synchronized walker and accumulator threading scheme. Wire frame arrows indicate asynchronous calls.

Manually setting thread affinity and using the simultaneous multi-threading (SMT) feature. We noticed that the code performs better significantly. We used the GNU interface to the POSIX scheduling. As we executed the new code with 1 thread per physical core, we set the affinity of each thread to 4 contiguous, non-overlapping virtual cores, except for the master thread that shares the same affinity with one of the worker threads in the pool, as they do not run concurrently.

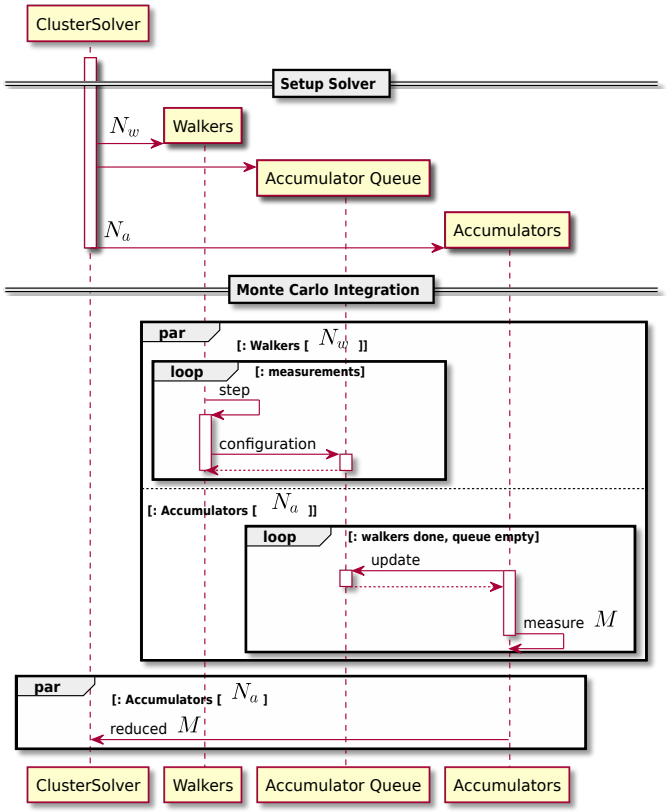

Fig. 4. UML sequence diagram of separate walker and accumulator threads. Wire frame arrows indicate asynchronous calls.

The speedup of the new code due to these settings is reported in Table I. The old code exhibits speedup as well when using the `smt 4` flag, hence we use it for all the subsequent measurements.

TABLE I
SPEEDUP OF THE NEW CODE DUE TO CHOICE OF SIMULTANEOUS MULTI-THREADING (SMT) LEVEL, AND DUE TO THE MANUAL BINDING OF THE THREAD AFFINITY. THE SAME NUMBER OF THREADS EQUAL TO THE NUMBER OF PHYSICAL CORES HAS BEEN USED.

	SMT = 1	SMT = 4
default affinity	-	1.5×
manual affinity	1.07×	2×

Custom thread pool. DCA version 1.1.0 implemented multi-threading by continuously spawning and merging POSIX threads. This induced a large overhead that was compounded by the fact that (3) was parallelized over the inner summation of $\vec{k}$ points, rather than on the cluster points $\vec{K}$ and frequencies ω_n. This caused thousands of threads to be spawned and joined.

Besides a better parallelization scheme, we also need a thread pool to limit the creation of OS threads in DCA++, especially for inputs that require running the coarse-graining multiple times. We implemented a custom thread pool using C++11 `std::thread`. We maintain an array of `std::thread` objects and array of queues of work

TABLE II
NEW CONTRIBUTIONS IN DCA 2.0 OVER DCA 1.1.0.

Implementation	**DCA 1.1.0**	**DCA 2.0**
Threading model	POSIX threads with short lifespan	Customized thread pool using C++ `std::thread`
Coarse-graining	MPI with poor intra-node work distribution	• MPI and improved multi-threading • Improved intra-process scheduling • Specialized implementations for inversion of small matrices • Reduced floating-point operations by using spin symmetry
Monte Carlo walker computation	On GPU	On GPU, with improved asynchronous CPU-GPU communications using multiple GPU's on Summit and NVLink
Monte Carlo accumulator computation	On CPU	On GPU – this is the major contribution to the performance improvement
Walker-accumulator interaction	• Single accumulator queue with spin locks • Temporary configuration copies on the CPU • Static workload distribution	• Improved implementation: avoids spinning on locks using conditional variables • Direct GPU-GPU communication between walker and accumulator • Dynamic intra-node workload distribution • Overlapping computation and communication

items represented by `std::packaged_task` objects. The completion of an asynchronous call can be queried through the generated `std::future` object. The work is dispatched in a simple round-robin fashion, with an additional integration of a work-stealing algorithm. The same thread pool is used both in the coarse-graining step and in the Monte Carlo integration, where further synchronization between walker and accumulator threads is expressed in terms of `std::mutex` and `std::conditional_variable` objects.

Compared to the naked POSIX implementation, we have significantly lowered the overhead. A clean trace can be generated by most CPU profilers, as the number of OS threads is constant during the execution. We have obtained a significant performance improvement by being able to access the underlying POSIX implementation on Summit, and manually setting the thread affinity, as shown above.

E. New contributions in DCA++ 2.0 over DCA++ 1.1.0

Table II summarizes the new contributions in DCA++ 2.0 over the older DCA++ 1.1.0 code that lead to the performance improvements described in this paper.

IV. RESULTS

A. System Architecture: Summit

For our evaluation we mainly used OLCF's Summit supercomputer. Summit is a 200 PFLOPS IBM AC922 system that was ranked the first place in the TOP500 list in June 2019 [1]. It delivers approximately 8 times the computational performance of Titan with 4608 hybrid nodes. Each Summit node contains 2 IBM POWER9 22C 3.07GHz CPUs with 512GB DDR4 RAM and 6 NVIDIA Volta V100 GPUs with 96GB high bandwidth memory (HBM) (divided into 2 sockets), all connected together with NVIDIA's high-speed NVLink.

B. System Setup

We ran the test case (described in section III-A) on Summit and compared the performance of the old version 1.1.0[4] and the new version 2.0[5] of the code. For each version, we used 1 GPU and 7 CPU cores per MPI rank - an even division of all 42 CPU cores on a node into 6 ranks. We ran 1 MPI rank per resource set[6], and 6 resource sets per node. We compiled the code with GCC 6.4 using optimization flags `-Ofast -funroll-loops -DNDEBUG -mcpu=power9 -mtune=power9`, and optimized the number of software threads for each version.

C. Application Setup

The old code version 1.1.0 was based on version 1.0.0 with minimal modifications to read the same input file and compute the same G_{tp} entries for a fair comparison with the new version. The structure of the code remains the same. We ran it with 13 accumulators and one walker per rank. It is because the accumulation took place on the CPU, which created a bottleneck in the process. Up to 16 accumulators can fit into the memory, but we observed no run-time benefit in mapping more accumulator threads to the same physical core.

We ran the new version of the code with 7 walkers and 7 accumulators, with each walker and accumulator sharing a

[4] https://github.com/CompFUSE/DCA/releases/tag/paper.2019.old_code
[5] https://github.com/CompFUSE/DCA/releases/tag/paper.2019.new_code
[6] The concept of "resource set" is introduced by IBM's job launcher `jsrun` developed for the Summit Power system.

thread in a MPI process (see Fig. 3). Since the MC walkers perform most of the compute intensive work, the run-time of the MC walkers is limited by the shared GPU resources. On the other hand, the run-time dependence on the number of accumulators is negligible. It is possible to use only one accumulator on a single thread to gather information from all MC walkers on other threads without a performance penalty. That is, the walker and the accumulator do not share the same thread (see Fig. 4).

D. Evaluation

1) ***Strong scaling performance****:* We first define the time-to-solution (TTS) as the "figure of merit" (FOM) to quantify the strong scaling performance of the code. This is equivalent to the time required to obtain a fixed number of MC measurements on different number of nodes. Fig. 5 reports the run-time comparison of the two versions performing a fixed number of 80 million measurements.

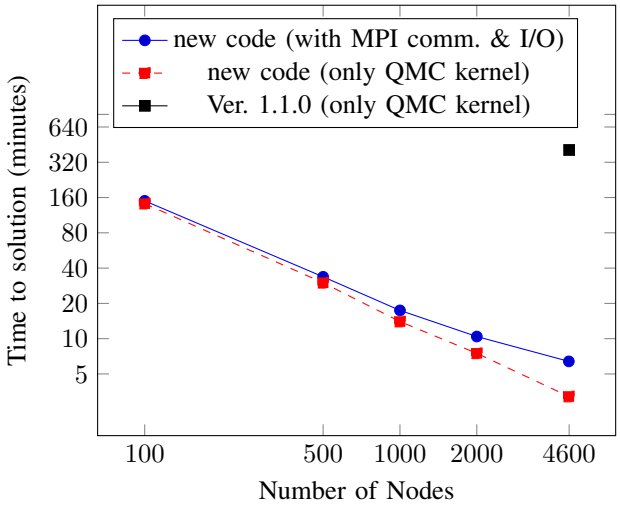

Fig. 5. Strong scaling plot (log-log) of a production run on OLCF's Summit, 80 million measurements (see Section III-A for the detailed system information). *Red squares* exhibit a linear scaling of the QMC kernel. *Blue circles* show the scaling of the entire DCA++ run-time including MPI communications for data movement across nodes and I/O from and to the GPFS file system on Summit. The *black square* shows the time spent (6.79 hours) by the old code on all 4600 nodes. We observe a performance improvement of up to 113× comparing the new code with the old code.

We first observe a significant speedup of 113× compared to the old code (version 1.1.0). This difference is mainly due to the asynchronous accumulation of measurements on the GPU in the new code.

With the new code, we observe that the Monte Carlo time exhibits a perfect strong scaling with the increasing node count (red squares). It is because as the number of measurements is evenly divided among the MC walkers, and the number of MC walkers is proportional to the number of nodes employed, the linear decrease in the QMC run-time is a direct consequence of the principle of division of work.

To understand the effect of I/O and communications overhead on the TTS, we plotted the total run-time (blue circles) as a comparison. The overhead is almost constant, around 3.5 minutes, for each node count. This is caused by the reading of the initial configuration from the parallel file system, the communication and averaging of the Monte Carlo results and in a small part by the coarse-graining. Although the coarse-graining is embarrassingly parallel across momentum and frequency points, we observed that the communication time can exceed the computation time. We therefore decided to limit the parallelism of this kernel to gangs of 100 nodes. To lower the overhead for global sum reductions, we pack different messages together, casting complex to real pairs and integer metadata to real numbers whenever it is possible. While the overhead is still significant, the time-to-solution is still better than the one reported in [9]. Improvements to the parallel file system can alleviate the overhead problem.

2) ***Weak scaling performance****:* Next, we performed a weak scaling analysis by increasing the number of measurements with the number of compute nodes, while keeping the run-time fixed at 13 minutes. In statistics, the standard error of the mean is inversely proportional to the square root of the number of measurements (i.e., $\sim 1/\sqrt{n_{\text{measurements}}}$). It is thus a reasonable FOM to verify the performance of the weak scaling for Monte Carlo simulations, which quantifies the improvements in the quality and precision of the simulations as a function of computing resources.

The error bars on G and G_{tp} were computed using the jackknife technique, with the measurements from one MPI rank grouped together as a bin. Fig. 6 reports the measured relative errors, which is the l_2 norm of the error normalized by the norm of the signal. As observed from the figure, the error scales inversely proportional to the square root of the number of measurements, which signifies a perfect weak scaling.

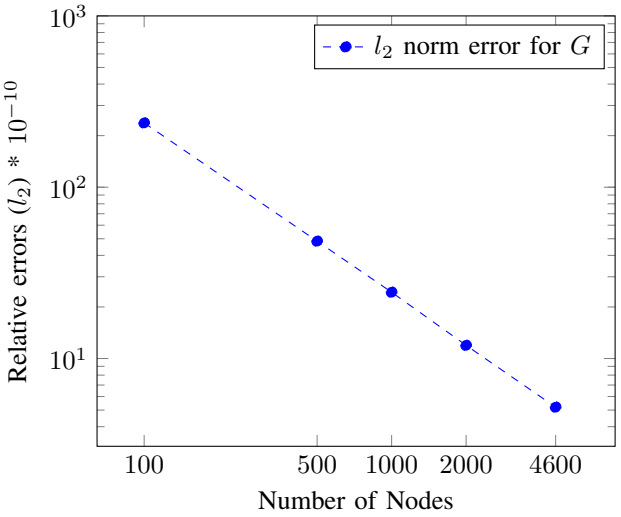

Fig. 6. Weak scaling plot (log-log) using the same system as in Fig. 5. For this plot, we fix the QMC run-time at 10 minutes and observe the number of measurements obtained. The higher the measurements, the higher is the accuracy, which leads to a lower error. Lower is better. This plot shows the error for G.

Another observation from Fig. 6 is that given the number of

measurements, the statistical error for G is much larger than the single precision accuracy. This hints at the possibility of making use of mixed precision to improve the performance: MC walks (Section III-C1) and the single-particle measurements (Section III-C2(c)) in double precision for the accuracy, while the two-particle measurements (Section III-C2(d)) are in single precision. Performance gained were measured as FLOPS count improvement in the following sub-subsection.

*3) **FLOPS count**:* We calculated the performance by updating a FLOP counter at each large matrix multiplication on a GPU in the walker routine and the G_{tp} accumulation routines. This provides a tight lower bound to the FLOPS count of our application. We validated our counting mechanism using the NVProf profiler on a single node. Considering the peak performance by the MC step, we got 64.1 PFLOPS for the new code, while the old code ran only at 0.577 PFLOPS as it was limited by the long run-time spent on the two-particle accumulation. With the mixed precision implementation, the new code achieved a peak performance of 73.6 PFLOPS on Summit. The higher FLOPS count is due to the use of single precision in the two-particle measurements, which is about two times faster than double precision operations. As the relative error on G_{tp} is significantly larger than 10^{-7} on our largest run, higher precision is not warranted for this calculation.

*4) **Roofline plot**:* In this section we analyze the most time-consuming kernels of our application. We focus on the Monte Carlo integration part, as the coarse-graining part is no longer relevant in terms of compute time or FLOPS after applying the optimizations presented in the previous section, at least for the chosen physical system.

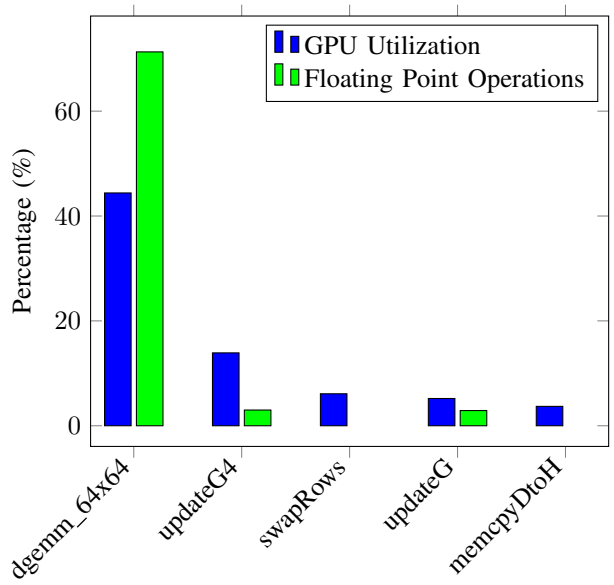

Fig. 7. Percentage of the GPU utilization and the floating point operations in each kernel and the memory transfer. *dgemm_64x64*, *swapRows* and *memcpyDtoH* (copy from GPU to CPU) are used by the MC walker, *updateG* is used by the single particle accumulator, and *updateG4* by the two-particle accumulator.

Fig. 7 reports the proportion of GPU utilization and floating point operations for each kernel contributing the most to the TTS, while the roofline plot in Fig. 8 shows the efficiency of the most FLOP-intensive kernels. These measurements were obtained using NVIDIA visual profiler (NVProf) on a serialized execution, i.e. one walker, one accumulator, and one GPU stream. We have validated our plot with the Intel Advisor on the PizDaint system[7] at Swiss National Supercomputing Centre (CSCS), ETH Zurich.

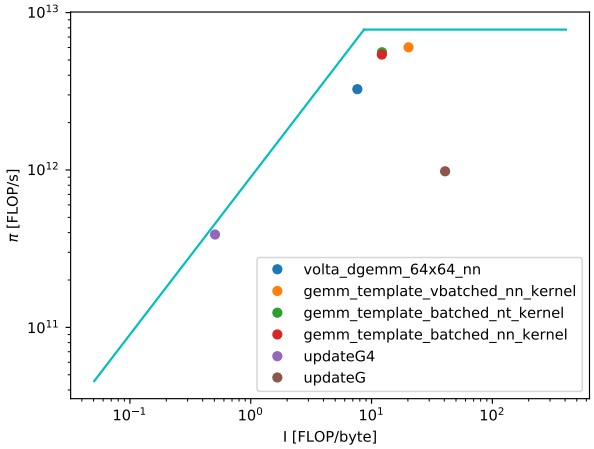

Fig. 8. Performance of the most FLOP-intensive kernels compared against the roofline model of an NVIDIA V100 GPU with a peak performance of 7.2 TFLOPS and a bandwidth of 900 GB/s. The legend is ordered by total FLOP consumption. The batched *gemm* kernels are part of the prepossessing in the two-particle accumulation, but they do not contribute significantly to the run-time.

Most of the time was spent on the *dgemm* kernel for matrix-matrix multiplications used by the MC walker, and on the *updateG4* kernel used by two-particle accumulation. One should note that these kernels are already highly optimized and close to the theoretical peak performance. The bandwidth of the kernel *swapRows*, used for the reordering of the MC configurations, is limited by the strided access to column-major data, and a similar *swapColumns* kernel is also necessary. In principle, the kernel *updateG* used in single-particle accumulation could be further optimized by distributing the workload among threads that would result in fewer bank conflicts. But given its small share in the run-time, the benefit of such an effort would be rather small.

In summary, the high performance of our application is attributed by a number of factors. Firstly, a good portion of the run-time is spent on efficient matrix-multiplications in the MC walkers that accounts for most of the FLOP count. Secondly, the overlap of computations and data transfers is enabled by using multiple GPU streams to keep the device busy during memory transfers and CPU updates. Finally, the parallelized measurement scheme contributes only a small overhead while greatly improves the concurrency.

[7]A Cray XC50 machine with Xeon E5-2690v3 12C 2.6GHz CPUs and NVIDIA Tesla P100 GPUs. Sixth place in the TOP500 list as of June 2019 [1].

V. Lessons learned

From our experience porting DCA++ to Summit, we outline a few good practices in software engineering (especially for C++ codes) which result in more readable, manageable, and reliable software:

- Develop good templated APIs for the bookkeeping of data structures and layout for the communication of arrays either over MPI or between CPUs and GPUs.
- Maintain an object-oriented design when possible. In our implementation, each computation step is represented by an object. Each object can be specialized both with an easily testable CPU-only implementation, and an optimized GPU-accelerated version. Usually the GPU specialization inherits from the CPU version. Template arguments are useful for selecting the implementation while maintaining the same API.
- Object encapsulation and memory re-utilization can be combined with a careful use of shared pointers to GPU resources.
- Each object is tested individually by unit tests and the CPU implementation acts as a baseline.

On the utilization of Summit, there are a few points to note apart from ordinary best practices on accelerator-based HPC:

- One needs to be hardware-aware and decide carefully how software threads are mapped to hardware threads. This includes the distribution of the CPU and GPU resources on a node into different resource sets, thread binding, and the simultaneous multi-thread (SMT) levels.
- With powerful GPUs, allowing multiple CPU threads, or MPI ranks, to access the same GPU can help increase the compute intensity. In this case, the GPU Multi-Process Service (MPS) should be used.
- Collective MPI reductions incur significant overhead on such a large machine. Minimize the number of calls to the MPI API by packing the data, and consider local re-evaluations instead of communications for relatively small kernels.
- Familiarize with the job scheduling system and use the appropriate job submission options (BSUB) and JSRUN flags[8] to allocate resources on Summit, in order to optimize utilization of the computing power.
- The performance tools (either vendor-developed or third parties) to show a holistic view of the CPU / GPU activities are still not mature enough on Summit. It was a challenge to optimize our code without having such information.

VI. Conclusion

In this paper we presented our new and improved algorithms for the DCA++ application (version 2.0). We evaluated and compared our latest improvements with the old version 1.0.0 on OLCF's Summit supercomputer. We observed a peak performance of 73.5 PFLOPS for the quantum Monte Carlo solver on Summit and up to $113\times$ performance improvement over the old code running at full scale on 4600 Summit nodes. The improved calculation of dynamical properties using modern programming models, to exploit the underlying hardware memory hierarchy, will provide important tests of the simplified models to explain real materials.

[8]One may use the "jsrunVisualizer" tool provided by OLCF (https://jsrunvisualizer.olcf.ornl.gov) to visualize the effects of jsrun options on the resource allocations on a Summit node.

Acknowledgment

Authors would like to thank Oscar Hernandez (ORNL), Jeff Larkin (NVIDIA), Don Maxwell (ORNL), Ronny Brendel (Score-P), John Mellor-Crummey (HPCToolkit) for their insights during the optimization phase of DCA++. This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences, Division of Materials Sciences and Engineering. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] "TOP500 List," 2018. [Online]. Available: https://www.top500.org/

[2] T. Maier, M. Jarrell, T. Pruschke, and M. H. Hettler, "Quantum cluster theories," *Rev. Mod. Phys.*, vol. 77, pp. 1027–1080, Oct 2005. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.77.1027

[3] M. H. Hettler, A. N. Tahvildar-Zadeh, M. Jarrell, T. Pruschke, and H. R. Krishnamurthy, "Nonlocal dynamical correlations of strongly interacting electron systems," *Phys. Rev. B*, vol. 58, pp. R7475–R7479, Sep 1998. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.58.R7475

[4] M. H. Hettler, M. Mukherjee, M. Jarrell, and H. R. Krishnamurthy, "Dynamical cluster approximation: Nonlocal dynamics of correlated electron systems," *Phys. Rev. B*, vol. 61, pp. 12 739–12 756, May 2000. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.61.12739

[5] P. W. Anderson, "The resonating valence bond state in la2cuo4 and superconductivity," *Science*, vol. 235, no. 4793, pp. 1196–1198, 1987.

[6] F. C. Zhang and T. M. Rice, "Effective hamiltonian for the superconducting cu oxides," *Phys. Rev. B*, vol. 37, pp. 3759–3761, Mar 1988. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.37.3759

[7] M. Troyer and U. Wiese, "Computational Complexity and Fundamental Limitations to Fermionic Quantum Monte Carlo Simulations," *Physical Review Letters*, vol. 94, no. 17, pp. 170 201–170 201, 2005.

[8] A. Georges, G. Kotliar, W. Krauth, and M. J. Rozenberg, "Dynamical mean-field theory of strongly correlated fermion systems and the limit of infinite dimensions," *Rev. Mod. Phys.*, vol. 68, pp. 13–125, Jan 1996. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.68.13

[9] M. S. Summers, G. Alvarez, J. Meredith, T. A. Maier, and T. Schulthess, "DCA++: A case for science driven application development for leadership computing platforms," *Journal of Physics: Conference Series*, vol. 180, pp. 012 077–012 077, 2009.

[10] G. Alvarez, M. S. Summers, D. E. Maxwell, M. Eisenbach, J. S. Meredith, J. M. Larkin, J. Levesque, T. A. Maier, P. R. C. Kent, E. F. D'Azevedo, and T. C. Schulthess, "New algorithm to enable 400+ tflop/s sustained performance in simulations of disorder effects in high-tc superconductors," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 61:1–61:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413433

[11] E. Gull, P. Werner, O. Parcollet, and M. Troyer, "Continuous-time auxiliary-field monte carlo for quantum impurity models," *EPL (Europhysics Letters)*, vol. 82, no. 5, p. 57003, 2008. [Online]. Available: http://stacks.iop.org/0295-5075/82/i=5/a=57003

[12] E. Gull, P. Staar, S. Fuchs, P. Nukala, M. S. Summers, T. Pruschke, T. C. Schulthess, and T. Maier, "Submatrix updates for the continuous-time auxiliary-field algorithm," *Phys. Rev. B*, vol. 83, p. 075122, Feb 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.83.075122

[13] P. Staar, T. A. Maier, and T. C. Schulthess, "Efficient non-equidistant fft approach to the measurement of single- and two-particle quantities in continuous time quantum monte carlo methods," *Journal of Physics: Conference Series*, vol. 402, no. 1, p. 012015, 2012. [Online]. Available: http://stacks.iop.org/1742-6596/402/i=1/a=012015

[14] U. R. Hähner, G. Alvarez, T. A. Maier, R. Solcà, P. Staar, M. S. Summers, and T. C. Schulthess, "DCA++: A software framework to solve correlated electron problems with modern quantum cluster methods," *Computer Physics Communications*, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465519300086

[15] E. Gull, P. Staar, S. Fuchs, P. Nukala, M. S. Summers, T. Pruschke, T. C. Schulthess, and T. Maier, "Submatrix updates for the continuous-time auxiliary-field algorithm," *Phys. Rev. B*, vol. 83, p. 075122, Feb 2011. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevB.83.075122

[16] N. J. Higham, "Stability of a method for multiplying complex matrices with three real matrix multiplications," *SIAM Journal on Matrix Analysis and Applications*, 1992. [Online]. Available: https://epubs.siam.org/doi/10.1137/0613043

[17] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

APPENDIX

Here we summarize the definition of all the relevant physical quantities used in this paper:

(a) $\Sigma_{\mathbf{c}}(\vec{K}, i\omega_n)$**:** cluster self energy. Either an input or the result of the previous DCA iteration. It represents the correction to the electron states due to interactions.

(b) $\Sigma_{\mathbf{DCA}}(\vec{k}, i\omega_n)$**:** Piecewise extension of $\Sigma_{\mathrm{c}}(\vec{K}, i\omega_n)$.

(c) $G(\vec{k}, i\omega_n)$ **or** G**:** single-particle Green's function. It describes the configuration of single electrons.

(d) $\bar{G}(\vec{K}, i\omega_n)$**:** coarse-grained single-particle Green's function.

(e) $\mathcal{G}_0(\vec{K}, i\omega_n)$ **or** $\mathcal{G}_0$**:** bare single-particle Green's function. It is computed from $\Sigma_c(\vec{K}, i\omega_n)$ and $\bar{G}(\vec{K}, i\omega_n)$ and is used as an input for the MC solver. It represents the configuration of non-interacting electrons.

(f) $G_{\mathbf{tp}}$**:** two-particle Green's function, also called the four-point function. Output of the MC solver used to determine relevant physical properties of the system. It represents the correlation between a pair of electrons.

(g) β**:** inverse temperature. The lower the temperature, the stronger the correlations between electrons are, and the higher the average expansion order in the MC integration is, directly impacting the run-time.

(h) H_0**:** non-interacting Hamiltonian. Used to compute the coarse-grained Green's function $\bar{G}(\vec{K}, i\omega_n)$. It represents the energy of a non-interacting electron with a given momentum.

(i) $H_{\mathbf{int}}$**:** interacting Hamiltonian. Input to the MC solver. It represents the interaction strength of pair of electrons on two given orbitals.

(j) k**:** expansion order of (6). Its average value scales as $O(\beta N_c U)$, where N_c is the cluster size and U is the average interaction strength. Not to be confused with the module of a momentum vector $\vec{k}$.

(k) M**:** $k \times k$ matrix. It is an intermediate result closely related to G stored by the single-particle accumulator.

(l) $N^{\sigma}_{\{s_i,\tau_i\}_k}$ **or** N**:** $k \times k$ matrix. Closely related to M and stored in the MC walker.

For the physical parameters chosen in this paper, all the single-particle function, i.e. G, $\mathcal{G}_0$ and Σ_{c} are ~5 MB large, while the two-particle Green's function $G4$ is 3.4 GB. The size of the matrices N and M fluctuates with the expansion order, but on average they are ~100 MB large.